

# In-Place Longest Common Extensions

Nicola Prezza\*

Technical University of Denmark, DTU Compute  
npree@dtu.dk

---

## Abstract

Longest Common Extension (LCE) queries are a fundamental sub-routine in several string-processing algorithms, including (but not limited to) suffix-sorting, string matching, compression, and identification of repeats and palindrome factors. A LCE query takes as input two positions  $i, j$  in a text  $T \in \Sigma^n$  and returns the length  $\ell$  of the longest common prefix between  $T$ 's  $i$ -th and  $j$ -th suffixes. In this paper, we present the following result: we can replace the (plain) text with a data structure of the *exact same size*— $n \lceil \log_2 |\Sigma| \rceil$  bits—supporting text extraction in optimal time and LCE queries in logarithmic time—i.e. *exponentially* faster than what can be achieved using the plain text alone. Our structure can be built in  $\mathcal{O}(n \log n)$  expected time and linear space. We show that our result is a powerful tool that can be used to efficiently solve in-place a wide variety of string processing problems: we provide the first in-place algorithms to compute the LCP array in  $\mathcal{O}(n \log n)$  expected time (the previous fastest in-place algorithm runs in  $\mathcal{O}(n^2)$  time) and to suffix-sort—with high probability of success—any set of  $b$  text suffixes in  $\mathcal{O}(n + b \log^2 n)$  expected time (the previous fastest in-place algorithm runs in  $\mathcal{O}(nb)$  time).

**1998 ACM Subject Classification** E.1 DATA STRUCTURES

**Keywords and phrases** Longest Common Extensions, in-place, LCP array, sparse suffix sorting

**Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.23

## 1 Introduction

Let  $T$  be a length- $n$  text over an integer alphabet  $\Sigma = \{0, \dots, \sigma - 1\}$ . In this paper, we consider the *longest common extension* (LCE) problem: to efficiently support (with some data structure) the computation of the length  $\ell$  of the longest common prefix between any two  $T$ 's suffixes. This problem finds important applications which include (but are not limited to): suffix-sorting, computing palindrome factors [1, 6], identification of repeats [22, 24], and string matching [2, 25, 29].

Clearly, constant-time random access to  $T$  is sufficient to support  $\mathcal{O}(\ell)$ -time LCE queries by direct comparison of the two suffixes and is also time-efficient in the average case as the longest common prefix between any two  $T$ 's suffixes does not exceed expected length  $\mathcal{O}(\log_\sigma n)$  on uniform texts. However, as the repetitiveness of  $T$  increases,  $\ell$  can be as large as  $n$  so more time-efficient solutions are required. Constant-time LCE queries are possible at the expenses of space usage. One way to achieve this goal is to build a  $\mathcal{O}(1)$ -time lowest common ancestor structure (see, e.g., [14]) on the suffix tree of the text (ST+LCA). A second solution consists in combining the longest common prefix array, a  $\mathcal{O}(1)$ -time range minimum query structure (see, e.g. [8]), and the inverted suffix array (RMQ+LCP). Both solutions require  $\mathcal{O}(n \log n)$  bits of working space, so for very large texts are not feasible in practice. One way to space-efficiently solve this problem is to flank the text  $T$  with some (sublinear-space)

---

\* Part of this work was done while the author was a PhD student at the University of Udine, Italy. Work supported by the Danish Research Council (DFF-4005-00267)



© Nicola Prezza;  
licensed under Creative Commons License CC-BY

42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

data structure. In their recent works [4, 5], Bille et al. follow this approach and propose a number of solutions to this problem that trade query-time for space-usage. Their most practical solution is a Monte Carlo data structure requiring  $\mathcal{O}(nw/\tau)$  bits on top of the text,  $w$  and  $0 < \tau \leq n$  being the memory word size and the block size, respectively, and answering LCE queries w.h.p. (with high probability) in  $\mathcal{O}(\tau)$  time. This structure can be built in  $\mathcal{O}(n \log(n/\tau))$  time. They also propose deterministic and Las Vegas structures with the same space-time bounds, but their preprocessing times— $\mathcal{O}(n^{2+\epsilon})$  and  $\mathcal{O}(n^{3/2})$ —make them not practical on big input texts. Deterministic data structures with similar time bounds but considerably improved construction times (of  $\mathcal{O}(n\tau)$ ) have been recently described by Tanimura et al. in [32]. While using a negligible amount of space on top of the text (for big enough  $\tau$ ), these solutions are still not able to reach the information-theoretic minimum amount of space of  $n \log \sigma$  bits while supporting fast queries at the same time.

**Our contributions** In this work, we obtain the following result: in the RAM model with word size  $w$ ,  $n \lceil \log \sigma \rceil$  bits of space are sufficient to answer *deterministic*  $\mathcal{O}(\log^2 \ell)$ -time LCE queries and optimal  $\mathcal{O}(m \log \sigma / w)$ -time extraction of any length- $m$  text substring (all logarithms are in base 2). In the case the alphabet is composed by  $k$ -bits integers, this space is  $nk$  bits: *exactly* the information-theoretic minimum number of bits required to represent any  $T \in \Sigma^n$ . LCE query times can be improved to  $\mathcal{O}(\log \ell)$  by adding  $\mathcal{O}(\log n)$  memory words to the space usage (which is negligible in practice). Despite not being directly applicable to our case, a recent work [23] suggests that our query times could be close to optimal within this space (the optimal would be  $\mathcal{O}(\log_\sigma n)$ , but the result in [23] applies only to sub-linear structures built on top of the text). Our data structure is based on Karp-Rabin fingerprinting and can be built in  $\mathcal{O}(n \log n)$  expected time and  $\mathcal{O}(n)$  words of space. In order to achieve this result, we introduce new techniques of general interest, including a statistical compression technique and several de-randomization procedures for Karp-Rabin fingerprinting. We point out that our result is a powerful tool that can be used to solve in-place a wide range of string-processing problems. To prove this statement, we present the first practical in-place Longest Common Prefix (LCP) array construction algorithm. Our algorithm runs in  $\mathcal{O}(n \log n)$  expected time and uses  $\mathcal{O}(1)$  memory locations on top of  $T$  and the LCP array. The previous fastest in-place LCP array construction algorithm [26] runs in quadratic time. As intermediate result, we describe a Monte Carlo LCE data structure taking  $n \lceil \log \sigma \rceil$  bits of space and admitting an in-place construction algorithm running in  $\mathcal{O}(n)$  expected time: the construction algorithm replaces the text with the data structure while using  $\mathcal{O}(1)$  words of extra working space. We apply this result to the suffix sorting problem and obtain an in-place Monte Carlo algorithm to suffix-sort any subset of  $b$  text suffixes with high probability in  $\mathcal{O}(n + b \log^2 n)$  expected time. This is the first in-place solution (with non-trivial running times) for the *sparse suffix sorting* problem, and improves the space of the state of the art [3, 9, 12, 16, 20].

In Table 1 we give an overview of the main solutions for the LCE problem described in the literature, distinguishing between Monte Carlo (first three lines), and deterministic structures.

## 2 Preliminaries

We assume our input to be a rewritable text of length  $n$  drawn from an integer alphabet  $\Sigma = \{0, \dots, \sigma - 1\}$  stored using  $\lceil \log \sigma \rceil$  bits per character.  $w \geq 1$  is the memory word size (in bits). Since we will make use only of integer additions, multiplications, modulo,

Space (bits)	Query time	build time	Notes	Reference
$n\lceil\log\sigma\rceil + \mathcal{O}(nw/\tau)$	$\mathcal{O}(\tau)$	$\mathcal{O}(n\log(n/\tau))$	$\tau \leq n$ , w.h.p.	[4]
$n\lceil\log\sigma\rceil + w\log n$	$\mathcal{O}(\log\ell)$	$\mathcal{O}(n)$ exp.	w.h.p.	Theorem 3
$n\lceil\log\sigma\rceil$	$\mathcal{O}(\log^2\ell)$	$\mathcal{O}(n)$ exp.	w.h.p.	Theorem 2
$\mathcal{O}(n\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	exact	<i>ST + LCA</i>
$\mathcal{O}(n\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	exact	<i>RMQ + LCP</i>
$n\lceil\log\sigma\rceil + \mathcal{O}(nw/\tau)$	$\mathcal{O}(\tau)$	$\mathcal{O}(n^{2+\epsilon})$	$\tau \leq n$ , exact	[4]
$n\lceil\log\sigma\rceil + \mathcal{O}(nw/\tau)$	$\mathcal{O}(\tau)$	$\mathcal{O}(n^{3/2})$ exp.	$\tau \leq n$ , exact	[4]
$n\lceil\log\sigma\rceil + \mathcal{O}(nw/\tau)$	$\mathcal{O}(\tau\log\tau)$	$\mathcal{O}(n\tau)$	$\tau \leq \frac{n}{\log n}$ , exact	[32]
$n\lceil\log\sigma\rceil + \mathcal{O}(w\log n)$	$\mathcal{O}(\log\ell)$	$\mathcal{O}(n\log n)$ exp.	exact	Theorem 7
$n\lceil\log\sigma\rceil$	$\mathcal{O}(\ell)$	—	exact	store only $T$
$n\lceil\log\sigma\rceil$	$\mathcal{O}(\log^2\ell)$	$\mathcal{O}(n\log n)$ exp.	exact	Theorem 7
$\mathcal{O}(zw\log n\log^*n)$	$\mathcal{O}(\log n\log^*n)$	$\mathcal{O}(n\log z)$	exact	[17, 30]

■ **Table 1** Main solutions for the LCE problem described in the literature. The first three solutions return the correct result *with high probability* (w.h.p.), while the others are exact. We account also for the space to store the plain text if this is required to answer LCE queries.  $\ell = LCE(i, j)$  is the output of the LCE query. In the first column, we omit any  $\mathcal{O}(w)$ -bits ( $\mathcal{O}(1)$  words) additional term. “exp.” after the build time (third column) means that construction times are in the expected case. If not specified, times are worst-case.  $z$  is the number of phrases of the LZ77 parsing without self-references.  $\log^* n$  is the iterated logarithm function.

and bitwise operations (masks, shifts), we assume that we can simulate a memory word of size  $w' = c \cdot w$  for any constant  $c$  with only a constant slowdown in the execution of these operations (see Appendix A for a description of how this can be achieved). In our proofs we assume  $n \geq 4$  (note that, if  $n < 4$ , then LCE queries can be trivially implemented in constant time),  $\log n \leq w$ , and  $\lceil\log\sigma\rceil \leq w$ . For the above consideration, our results will be easily generalizable to the case  $\log n \in \mathcal{O}(w)$  and  $\log\sigma \in \mathcal{O}(w)$ .

We will describe our ideas starting from the binary alphabet  $\sigma = 2$ . On more general alphabets, note that we can build a binary text  $T' \in \{0, 1\}^{n \cdot b}$ , where  $b = \lceil\log\sigma\rceil$ , by concatenating numbers  $T[0], T[1], \dots, T[n-1]$  written in binary and reduce the problem to the binary case as  $T.LCE(i, j) = \lfloor T'.LCE(i \cdot b, j \cdot b)/b \rfloor$ . This reduction will turn out to keep times alphabet-independent with our solution. To make notation more compact, with  $T[i, \dots, j]$  we denote both  $T$ 's substring starting at position  $i$  and ending at position  $j$ , and the integer with binary representation  $T[i]T[i+1] \dots T[j]$  (each  $T[i]$  being a  $\lceil\log\sigma\rceil$ -bits integer). If  $j < i$ ,  $T[i, \dots, j]$  denotes the empty string  $\epsilon$  or the integer 0. The use (string/integer) will be clear from the context. While computing the space taken by our data structures we drop any additive  $\mathcal{O}(w)$ -bits space term. For space constraints, we assume the reader to be familiar with Karp-Rabin fingerprinting [21].  $\phi_q : \{0, 1\}^* \rightarrow [0, q-1]$  indicates the Karp-Rabin hash function with modulo  $q$  on strings from the binary alphabet  $\{0, 1\}$ . W.h.p. (with high probability) means with probability at least  $1 - n^{-c}$  for an arbitrarily large constant  $c$ . If not otherwise specified, logarithms are in base 2.

### 3 Monte Carlo LCE data structure

Our general strategy to obtain a data structure supporting LCE queries follows the one described by Bille et al. [4, 5]. Our improvements over this result concern the space of our

structure and its construction time. First, we describe an optimal-space data structure supporting efficient computation of the Karp-Rabin fingerprint of any text substring. Our original idea is to *replace* the text (not just augment it) with the fingerprints of a subset of its prefixes. The loss in space efficiency is avoided by repeatedly picking the Karp-Rabin module  $q$  very close, but above, a power of two until all residues are below that power of two (thus saving 1 bit per stored fingerprint). As a result, we are able to achieve  $n \lceil \log \sigma \rceil$  bits of space usage for our structure. Then, to compute  $LCE(i, j)$  on  $T$  we perform the following two steps. (1) *Exponential search*: we compare  $\phi_q(T[i, \dots, i+2^k])$  and  $\phi_q(T[j, \dots, j+2^k])$  for  $k = 0, 1, \dots$  until the two compared strings differ. Letting  $\ell = LCE(i, j)$ , this phase terminates in  $\mathcal{O}(\log \ell)$  steps. (2) *Binary search*: we compare  $\phi_q(T[i, \dots, i+m])$  and  $\phi_q(T[j, \dots, j+m])$  for  $\mathcal{O}(\log \ell)$  values of  $m$  using binary search in the interval found in step (1) until we find the value  $\ell$  such that  $\phi_q(T[i, \dots, i+\ell-1]) = \phi_q(T[j, \dots, j+\ell-1])$  and  $\phi_q(T[i, \dots, i+\ell]) \neq \phi_q(T[j, \dots, j+\ell])$ . Follows a detailed description of our Monte Carlo data structure; in Section 4 we show how to de-randomize our solution so that it computes always the correct LCE.

### 3.1 Fingerprint Sampling

We start considering the binary case  $\sigma = 2$ , and then extend the result to more general alphabets. We introduce two sources of randomness in our structure: the *modulus*  $q$  and the *seed*  $\bar{s}$ . First, we choose a random prime  $q$  uniformly in the interval  $[2, 2^w - 1]$ . We define a *block size*  $\tau = \lceil \log q \rceil$ . Without loss of generality, we assume that  $n$  is a multiple of  $\tau$  (the general case can be reduced to this case by left-padding the text with  $\tau - (n \bmod \tau)$  bits). At this point, we choose uniformly a random number (the seed)  $\bar{s}$  in the interval  $[0, q - 1]$ .  $\bar{s}$  is a  $\tau$ -bits integer (after a suitable left-padding of zeros); we left-pad our binary text  $T$  with  $\bar{s}$  written in binary. Clearly,  $LCE(i, j)$  queries on  $T$  can still be solved using the padded text  $\bar{s}T$  by simply adding  $\tau$  to the arguments of LCE (i.e. solving  $LCE(i + \tau, j + \tau)$  on the padded text). To improve readability, in what follows we assume that  $T$  is prefixed by  $\bar{s}$  (and thus write just  $T$  and  $n$  instead of  $\bar{s}T$  and  $n + \tau$ , respectively).

Let  $B, P' \in [0, q - 1]^{n/\tau}$  be the integer arrays defined as

$$B[i] = T[i \cdot \tau, \dots, (i + 1) \cdot \tau - 1], \quad i = 0, \dots, n/\tau - 1$$

and

$$P'[i] = \sum_{j=0}^i 2^{(i-j) \cdot \tau} \cdot B[j] \bmod q = \phi_q(T[0, \dots, (i + 1) \cdot \tau - 1])$$

First, note that  $B[i] - q \leq (2^\tau - 1) - 2^{\tau-1} < 2^{\tau-1} \leq q$ , so  $\lfloor B[i]/q \rfloor \in \{0, 1\}$  holds. We build a bitvector  $D[0, \dots, n/\tau - 1]$  defined as  $D[i] = \lfloor B[i]/q \rfloor$ . To simplify notation, let  $P'[-1] = 0$ . At this point,  $B$ 's values can be retrieved as  $B[i] = (P'[i] - 2^\tau \cdot P'[i - 1] \bmod q) + D[i] \cdot q$ . Arrays  $P'$  and  $D$  take  $n + n/\tau$  bits of space and replace the text in that they support the retrieval of any  $B[i]$ . First, we show how to compute efficiently  $\phi_q(T[i, \dots, j])$  for any  $0 \leq i \leq j < n$  by using  $P'$  and  $D$ . We then show how to reduce the space usage to  $n$  bits while still being able to support constant-time text extraction and Karp-Rabin fingerprint computation. Let  $j = \lfloor i/\tau \rfloor$ . Then,

$$\begin{aligned} \phi_q(T[0, \dots, i]) &= \phi_q(T[0, \dots, j \cdot \tau - 1]) \cdot 2^{i-j \cdot \tau + 1} + \phi_q(T[j \cdot \tau, \dots, i]) \bmod q \\ &= P'[j - 1] \cdot 2^{i-j \cdot \tau + 1} + \lfloor B[j]/2^{\tau-i+j \cdot \tau - 1} \rfloor \bmod q \end{aligned}$$

We now have to show how to compute the fingerprint  $\phi_q(T[i, \dots, j])$ ,  $j \geq i$  of any text

substring. This can be easily achieved by means of the equality:

$$\phi_q(T[i, \dots, j]) = \phi_q(T[0, \dots, j]) - \phi_q(T[0, \dots, i-1]) \cdot 2^{j-i+1} \pmod{q} \quad (1)$$

Computing  $2^e \pmod{q}$  takes  $\mathcal{O}(\log e)$  time with the fast exponentiation algorithm, therefore the computation of  $\phi_q(T[i, \dots, j])$  takes  $\mathcal{O}(\log(j-i+1))$  time with our structure for all  $0 \leq i \leq j < n$ .

### 3.2 Reducing space usage

In order to remove the  $n/\tau$ -bits overhead, we build an auxiliary array  $S = \langle i : \lfloor P'[i]/2^{\tau-1} \rfloor = 1, i = 0, \dots, n/\tau - 1 \rangle$  storing all  $i$ 's such that the most significant bit of  $P'[i]$  is equal to 1. At this point, we replace  $P'$  with an array  $P$  of  $n/\tau$  integers of  $(\tau-1)$  bits each defined as  $P[i] = P'[i] \pmod{2^{\tau-1}}, i = 0, \dots, n/\tau - 1$ , i.e. we remove the most significant bit from each  $P'[i]$ .  $P$  takes  $n \cdot (\tau-1)/\tau$  bits of space. Clearly, by using  $P$  and  $S$  we can retrieve any  $P'[i]$  in  $\mathcal{O}(|S|)$  time with a simple linear scan on  $S$ . The main idea, at this point, is to choose the prime  $q$  in such a way that the expected size of  $S$  becomes constant. After achieving this goal, the overall space of  $P$ ,  $D$ , and  $S$  will be of  $n$  bits (plus a constant number of memory words) and we will still be able to retrieve any  $B[i]$  and  $P'[i]$  in constant time.

We reverse our strategy. We first choose a block size  $\tau \in \Theta(w)$ , pick a uniform prime  $q$  such that  $\lceil \log q \rceil = \tau$ , and then choose a uniform seed  $\bar{s}$  in  $[0, q-1]$ . In Sections 3.3 and 3.4 we show how to choose  $\tau$ . The key point is that each  $P'[i]$  is a uniform random variable taking values in the range  $[0, q-1]$ . To prove this statement, note that  $P'[i]$  can be written as  $P'[i] \equiv_q \bar{s} \cdot 2^{i\tau} + \bar{t}_i$ , where  $\bar{t}_i = \phi_q(T[\tau, \dots, (i+1) \cdot \tau - 1])$ . Let  $\mathcal{P}(P'[i] = x)$ ,  $x < q$ , be the probability that  $P'[i]$  is equal to  $x$ . Then, for any  $x < q$ ,

$$\mathcal{P}(P'[i] = x) = \mathcal{P}(\bar{s} \cdot 2^{i\tau} + \bar{t}_i \equiv_q x) = \mathcal{P}(\bar{s} \equiv_q (x - \bar{t}_i) \cdot 2^{-i\tau}) = 1/q \quad (2)$$

The fact that  $q$  is prime guarantees the existence of the inverse of  $2^{i\tau}$  modulo  $q$ . Let  $x < q$ . Equation 2 implies, in particular, that

$$\mathcal{P}(P'[i] < x) = x/q \quad (3)$$

Let  $\bar{1}_i \in \{0, 1\}$  be the indicator random variable taking value 1 iff the most significant bit of  $P'[i]$  is equal to 1. Equation 3 implies that  $\bar{1}_i$  has a Bernoullian distribution with success probability  $p = (q - 2^{\tau-1})/q$ . We want this probability to be at most  $1/n$  in order to get an expected constant size for  $S$ . By solving  $(q - 2^{\tau-1})/q \leq 1/n$  and by adding the constraint  $\lceil \log q \rceil = \tau$ , we obtain that the interval  $\mathcal{Z}$  from which we have to uniformly pick  $q$  in order to satisfy both constraints is

$$\mathcal{Z} = \left[ 2^{\tau-1}, 2^{\tau-1} \left( \frac{n}{n-1} \right) \right] \quad (4)$$

The issue to efficiently pick a uniform prime from an interval has been addressed in [18]. At this point,  $S$ 's expected size can be computed as

$$E[|S|] = E \left[ \sum_{i=0}^{n/\tau-1} \bar{1}_i \right] = \sum_{i=0}^{n/\tau-1} E[\bar{1}_i] = (n/\tau) \cdot E[\bar{1}_i] = (n/\tau) \cdot (q - 2^{\tau-1})/q \leq (n/\tau) \cdot (1/n) = 1/\tau \quad (5)$$

where  $E[X]$  is the expected value of random variable  $X$ . Recall that the expected value of a Bernoullian random variable with success probability  $p$  is  $p$ .

Let  $b = \lceil \log \sigma \rceil$ . On a general alphabet size such that  $b \leq w$ , the text is processed as a binary sequence of  $nb$  bits, and the validity of the above results is preserved by substituting  $n$  with  $nb$  in Equation 4. In particular,  $\mathcal{Z}$ 's size becomes  $|\mathcal{Z}| = 2^{\tau-1}/(nb-1) \geq 2^{\tau-1}/(nb) = 2^{\tau-1-\log n-\log b}$ . Since we assume  $w \geq \log n$  and  $w \geq b \geq \log b$ , we get the lower bound  $|\mathcal{Z}| \geq 2^{\tau-1-2w}$ .

**Number of primes in  $\mathcal{Z}$**  Let  $\pi(x)$  denote the number of primes smaller than  $x$ . Let moreover  $A = 2^{\tau-1}$  and  $H = 2^{\tau-1-2w}$  be the smallest element contained in  $\mathcal{Z}$  and the lower bound for  $|\mathcal{Z}|$  above computed, respectively. Our aim in this paragraph is to compute a lower bound for the number  $z_p = \pi(A+H) - \pi(A)$  of primes contained in  $\mathcal{Z}$ . This will be needed later in order to compute the collision probability of our hash function. Note that the Prime Number Theorem can be applied to solve this task only if  $H \geq A \cdot c$ , for some fixed  $c > 0$ , so we cannot use it in our case. Luckily for us, Heath-Brown [15] proved (see also [27]) that, if  $H$  grows at least as  $A^{7/12}$ , then  $\pi(A+H) - \pi(A) \sim H/\log_e A$  (for  $A \rightarrow \infty$ ,  $e$  is the natural logarithm base). Solving  $H \geq A^{7/12}$  we get the constraint

$$\tau \geq (24/5)w + 1 \quad (6)$$

Later we will show how to choose  $\tau$  (keeping (6) in mind). Heath-Brown's theorem gives us  $z_p \geq H/\log_e A = 2^{\tau-1-2w}/\log_e(2^{\tau-1}) \geq 2^{\tau-1-2w}/(\tau-1) \geq 2^{\tau-1-2w}/\tau$ .

### 3.3 Deterministic space

With the above strategy, our structure takes  $n \lceil \log \sigma \rceil$  bits of space with high probability only. We can make the space deterministic by picking multiple random pairs  $q, \bar{s}$  as described above and re-building the structure until this requirement is satisfied. Our goal in this section is to compute the expected number  $R$  of pairs  $q, \bar{s}$  we have to randomly pick before obtaining a constant size for  $S$  (in our calculations below, we focus on  $|S| = 0$ ). Note that  $E[|S|] = \sum_{k>0} k \cdot \mathcal{P}(|S| = k)$  and  $\mathcal{P}(|S| > 0) = \sum_{k>0} \mathcal{P}(|S| = k)$ , so  $\mathcal{P}(|S| > 0) \leq E[|S|]$  holds. From Equation 5,  $E[|S|] \leq 1/\tau$ , therefore  $\mathcal{P}(|S| > 0) \leq 1/\tau$ . This yields  $\mathcal{P}(|S| = 0) \geq 1 - 1/\tau$ . We choose

$$\tau = cw \geq c \log n \quad (7)$$

for any constant  $c \geq 1$ , so the above probability is at least  $1 - 1/\log n$ . Later we will show how to choose  $c$  keeping in mind also Constraint (6). Finally, since we assume  $n \geq 4$  (and therefore  $\log n \geq 2$ ), we obtain  $\mathcal{P}(|S| = 0) \geq 0.5$ . Given that we repeat the construction of our structure as long as  $|S| > 0$  holds, the number  $R$  of times we repeat the construction is a geometric random variable with success probability  $p = \mathcal{P}(|S| = 0) \geq 0.5$ , and has therefore expected value  $1/p \leq 2$ . Note that, since  $\tau \in \Theta(w)$  and  $\lceil \log \sigma \rceil \leq w$ , arrays  $P$  and  $D$  have  $\mathcal{O}(n)$  entries each. We obtain the following Lemma:

► **Lemma 1.** *In  $\mathcal{O}(n)$  expected time we can build arrays  $P$ ,  $D$ , and  $S$  taking overall  $n \lceil \log \sigma \rceil$  bits of space and supporting the computation of any  $B[i]$  and  $P'[i]$  in constant time.*

### 3.4 A Monte Carlo in-place LCE data structure

On a binary alphabet, we can easily answer  $LCE(i, j)$  by comparing  $\phi_q(T[i, \dots, i+k])$  with  $\phi_q(T[j, \dots, j+k])$  for  $\mathcal{O}(\log n)$  values of  $k$  with binary search. We can furthermore improve query times by performing an exponential search before applying the binary search procedure. We compare  $\phi_q(T[i, \dots, i+k])$  with  $\phi_q(T[j, \dots, j+k])$  for  $k = 2^0, 2^1, 2^2, \dots$  until the two

fingerprints differ. Letting  $\ell = LCE(i, j)$ , this procedure terminates in  $\mathcal{O}(\log \ell)$  steps. We then apply the binary search procedure described above on the interval of size  $\mathcal{O}(\ell)$  obtained with the exponential search. Each exponential and binary search step take  $\mathcal{O}(\log \ell)$  time (from the fast exponentiation algorithm).

On a more general alphabet, each character takes  $b = \lceil \log \sigma \rceil \in \mathcal{O}(w)$  bits, and our structure is therefore built over a binary text  $T'$  of length  $n \cdot b$ . We can make query times alphabet-independent as follows. First of all, while computing  $T.LCE(i, j)$  we perform exponential and binary searches by comparing  $\phi_q(T'[i \cdot b, \dots, (i+k) \cdot b])$  with  $\phi_q(T'[j \cdot b, \dots, (j+k) \cdot b])$ , i.e. we compare  $T'$  substrings starting and ending at character boundaries. This reduces the number of steps to be performed from  $\mathcal{O}(\log(\ell \cdot b))$  to  $\mathcal{O}(\log \ell)$ . At this point, note that each step requires the computation of  $2^{t \cdot b} \bmod q$  with the fast exponentiation algorithm,  $t \in \mathcal{O}(\ell)$  being the length of the two compared substrings ( $\mathcal{O}(\log(\ell \log \sigma))$  time). Since  $b$  is a common factor in all exponents, we can pre-compute  $Y = 2^b \bmod q$  and—at each step—compute  $Y^t \bmod q$  instead of  $2^{t \cdot b} \bmod q$  with the fast exponentiation algorithm. This reduces the number of steps of the exponentiation algorithm to  $\mathcal{O}(\log \ell)$ .

Finally, note that extracting text corresponds to reading array  $B$  ( $\tau \in \Theta(w)$  bits of the text per  $B$  element).

### 3.4.1 Wrong LCE probability

We start the analysis from the binary case  $\sigma = 2$ . Let  $C$  be the random variable denoting the number of pairs  $\langle X, Y \rangle$  of  $T$  substrings with  $|X| = |Y|$  that generate a collision, i.e.  $X \neq Y$  and  $\phi_q(X) = \phi_q(Y)$ . Our goal is to compute an upper bound for the probability  $\mathcal{P}(C > 0)$ . Clearly,  $\mathcal{P}(C > 0)$  is an upper bound to the probability of computing a wrong LCE with our structure. Let  $X_i^k$  denote  $T$ 's substring of length  $k$  starting at position  $i$ . There is at least one collision ( $C > 0$ ) iff  $X_i^k \equiv_q Y_j^k$  for at least one pair  $X_i^k \neq Y_j^k$ , i.e. iff  $q$  divides at least one of the numbers  $|X_i^k - Y_j^k|$  such that  $X_i^k \neq Y_j^k$ . Since  $q$  is prime, this happens iff  $q$  divides their product  $z = \prod_{k=1}^{n-1} \prod_{i,j: X_i^k \neq Y_j^k} |X_i^k - Y_j^k|$ . Since each  $|X_i^k - Y_j^k|$  has at most  $n$  binary digits and there are no more than  $n^2$  such pairs for every  $k$ , we have that  $z < 2^{n^4}$ . It follows that there cannot be more than  $n^4$  distinct primes dividing  $z$ .

Let  $b = \lceil \log \sigma \rceil$ . On a more general alphabet with  $b \leq w$ , each  $|X_i^k - Y_j^k|$  has at most  $nb$  binary digits, and we obtain  $z < 2^{n^4 \cdot b}$ . It follows that there cannot be more than  $n^4 \cdot b$  distinct primes dividing  $z$ . The probability of uniformly picking a prime  $q \in \mathcal{Z}$  dividing  $z$  is therefore upper bounded by  $n^4 \cdot b / z_p$ , where  $z_p$  is a lower bound on the number of primes contained in  $\mathcal{Z}$ , see Section 3.2. Recall that  $z_p \geq 2^{\tau-1-2w} / \tau$ , so  $n^4 \cdot b / z_p \leq n^4 \cdot b \cdot \tau / 2^{\tau-1-2w} = 2^{4 \log n + \log b + \log \tau + 1 + 2w - \tau}$ . We choose

$$\tau = (9 + c)w \tag{8}$$

for an arbitrarily large constant  $c$ . Being  $w \in \omega(1)$ , we assume<sup>1</sup>  $w \geq \log \tau = \log(9 + c) + \log w$ . We obtain  $\tau = (9 + c)w \geq (4 + c) \log n + \log b + \log \tau + 1 + 2w$ , therefore  $n^4 \cdot b / z_p \leq 2^{-c \log n} = n^{-c}$ . Note that this choice of  $\tau$  satisfies constraints (6) and (7). This leads to:

$$\mathcal{P}(\text{wrong LCE}) \leq \mathcal{P}(C > 0) \leq n^{-c}$$

For an arbitrarily large constant  $c$ . We obtain:

<sup>1</sup> Note that this inequality always holds after simulating a memory word of size  $w' = dw$  for a sufficiently large constant  $d$



► **Theorem 2.** *In  $\mathcal{O}(n)$  expected time we can build a data structure taking  $n\lceil\log\sigma\rceil$  bits of space and supporting extraction of any length- $m$  text substring and LCE queries w.h.p. in  $\mathcal{O}(m\log\sigma/w)$  and  $\mathcal{O}(\log^2\ell)$  worst-case time, respectively.*

### 3.4.2 Speeding up LCE queries

We consider a general alphabet size  $\sigma \leq 2^w$ . Let  $b = \lceil\log\sigma\rceil$  and let  $T' \in \{0,1\}^{n \cdot b}$  be the concatenation of  $T$ 's characters written in binary. We can avoid the overhead introduced by the fast exponentiation algorithm by pre-computing and storing (in  $\mathcal{O}(w\log n)$  bits) values  $z_i = 2^{b \cdot 2^i} \bmod q$ ,  $i = 0, \dots, \lceil\log n\rceil$  and performing binary search by splitting interval lengths in correspondence of powers of 2 as follows. First, note that  $z_0 = 2^b \bmod q$  and  $z_{i+1} = (z_i)^2 \bmod q$ , so the values  $z_i$  can be pre-computed in  $\mathcal{O}(\log n)$  time. Let the notation  $\langle i, j, e, k \rangle$ , with  $0 \leq i, j, e, k < n$  and  $e < k$ , denote that we already verified (w.h.p.) that  $T[i, \dots, i+e-1] = T[j, \dots, j+e-1]$  and  $T[i, \dots, i+k-1] \neq T[j, \dots, j+k-1]$ . We use this notation to indicate the state of a binary search step, and start from state  $\langle i, j, 0, n-j \rangle$  (we assume for simplicity that  $T[i, \dots, i+(n-j)-1] \neq T[j, \dots, n-1]$ ; otherwise,  $LCE(i, j) = n-j$ ). We use a modified version of Equation 1 by adding a parameter (exponential *exp*) to the Karp-Rabin hash function:

$$\phi'_q(T'[i, \dots, j], \text{exp}) = \phi_q(T'[0, \dots, j]) - \phi_q(T'[0, \dots, i-1]) \cdot \text{exp} \bmod q \quad (9)$$

Note that  $\phi_q(T'[i, \dots, j]) = \phi'_q(T'[i, \dots, j], 2^{(j-i+1) \cdot b})$ . At binary search step  $\langle i, j, e, k \rangle$  we still have to compare the last  $l = k - e$  characters of  $T[i, \dots, i+k-1]$  and  $T[j, \dots, j+k-1]$ . We split each of these two substrings suffixes in a left part of length  $l' = 2^{\lfloor \log(l/2) \rfloor}$  (i.e. the closest power of 2 smaller than or equal to  $l/2$ ) and a right part of length  $l - l'$ . Note that value  $2^{l' \cdot b} \bmod q = z_{\log l'} = z_{\lfloor \log(l/2) \rfloor}$  has been pre-computed, so we can compute and compare in constant time the two values

$$\phi_q(T'[(i+e) \cdot b, \dots, (i+e+l'-1) \cdot b]) = \phi'_q(T'[(i+e) \cdot b, \dots, (i+e+l'-1) \cdot b], z_{\lfloor \log(l/2) \rfloor})$$

and

$$\phi_q(T'[(j+e) \cdot b, \dots, (j+e+l'-1) \cdot b]) = \phi'_q(T'[(j+e) \cdot b, \dots, (j+e+l'-1) \cdot b], z_{\lfloor \log(l/2) \rfloor})$$

If the two values differ, then we recurse on  $\langle i, j, e, e+l' \rangle$ . If the two values are equal, then we recurse on  $\langle i, j, e+l', k \rangle$ . Note that we always compare (fingerprints of) strings whose lengths are powers of two. This will be crucial in the next section in order to efficiently de-randomize our structure. Since  $l/4 < l' \leq l/2$ , this binary search procedure terminates in  $\mathcal{O}(\log n)$  steps, each taking constant time. As done in the previous section, we can perform an exponential search before the binary search in order to reduce the size of the binary search interval from  $\mathcal{O}(n)$  to  $\mathcal{O}(\ell)$ . Note that with our sampling  $z_i$  it is straightforward to implement each exponential search step in constant time. We obtain:

► **Theorem 3.** *In  $\mathcal{O}(n)$  expected time we can build a data structure taking  $n\lceil\log\sigma\rceil + \mathcal{O}(w\log n)$  bits of space and supporting extraction of any length- $m$  text substring and LCE queries w.h.p. in  $\mathcal{O}(m\log\sigma/w)$  and  $\mathcal{O}(\log\ell)$  worst-case time, respectively.*

Note that values  $z_i$  need to be explicitly stored for the binary search as this step might need access to a  $z_i$  with arbitrarily small  $i$  (and, while we can quickly compute  $z_i$  from  $z_{i-1}$ , the opposite is not true). Note moreover that, even if we assumed  $\lceil\log\sigma\rceil \leq w$ , it is easy to see that our results are valid also for the more general case  $\log\sigma \in \mathcal{O}(w)$ : in the case alphabet characters are composed by  $\mathcal{O}(1)$  words, we can break them in a constant number of  $w$ -bits sub-characters. This introduces only a constant slowdown in the computation of LCE queries on the original text.



### 3.5 In-place construction algorithm

In this section we show that our data structure can be built *in-place*, i.e. we can replace the text with the data structure and use only  $\mathcal{O}(1)$  memory words of extra space during construction. A direct consequence of this fact is (see Section 5) the existence of an in-place algorithm to efficiently suffix-sort (with high probability of success) any subset of text positions.

We first consider the binary case  $\sigma = 2$ . First, we pick  $\tau, q, \bar{s}$  as described in the previous sections. We consider the text as a sequence  $B[0, \dots, n/\tau - 1]$  of integers in the range  $[0, 2^\tau - 1]$  (again, we assume for simplicity that  $\tau$  divides  $n$ ), and, for  $i = 0, \dots, n/\tau - 1$ , we replace the most significant bit of  $B[i]$  with the bit  $D[i]$ , and the remaining  $\tau - 1$  bits with the value  $P[i]$  (clearly, this allows to retrieve any  $P[i]$  and  $D[i]$  in constant time). If the most significant bit of  $P'[i]$  is equal to 1, then we append  $i$  to the vector  $S$  (initialized as empty). Note that  $P'[i]$ ,  $P[i]$ , and  $D[i]$  can all be easily computed in constant time by using  $P'[i - 1]$  and  $B[i]$  for  $i > 0$ , so we need to keep in memory only a constant number of memory words at each of the  $n/\tau$  steps. Let  $c$  be any constant chosen in advance. If the size of  $S$  exceeds  $c$  at some point of the construction, then we reverse the procedure and restore  $T$ . It is easy to see that  $B[i]$  can be computed in constant time by using  $P[i]$ ,  $P[i - 1]$ ,  $D[i]$ , and  $S$  (recall that  $B[i]$  has been replaced with  $P[i]$  and  $D[i]$  during construction), so we can restore the text in  $\mathcal{O}(n/w)$  time using  $\mathcal{O}(1)$  memory words of working space. After restoring the text, we empty array  $S$ .

For the same considerations made in the previous section, we need to repeat the construction at most  $\mathcal{O}(1)$  times in the expected case. At each round, we spend  $\mathcal{O}(n/w)$  time. On a general alphabet with  $\lceil \log \sigma \rceil \in \mathcal{O}(w)$ , we build the structure on the binary representation of the text and the construction algorithm terminates therefore in expected optimal  $\mathcal{O}(\frac{n \log \sigma}{w})$  time (assuming that the input text is already packed) while taking only  $\mathcal{O}(1)$  words on top of the space of the text.

## 4 Deterministic LCE data structure

The aim of this section is to show how to make the structure described in the previous section deterministic. We start by proving three theorems solving with different space/time tradeoffs the problem of checking whether  $\phi_q$  generates collisions over a specific subset of text substrings. Detailed proofs of Theorems 4-6 can be found in Appendix B.

► **Theorem 4.** *In  $\mathcal{O}(n \log n)$  expected time and  $\mathcal{O}(n)$  words of space we can check whether  $\phi_q$  is collision-free over all pairs of substrings of  $T$  having the same length  $k = 2^e$ , for all  $0 \leq e \leq \log n$*

**Proof.** (sketch) The idea is to iteratively check collisions between strings of length  $2^e$ , for  $e = 0, \dots, \log n$ . At step  $e$ , we hash text positions  $i = 0, \dots, n - 2^e$  according to the fingerprint of  $T[i, \dots, i + 2^e - 1]$  and exploit the results of step  $e - 1$  to check string equality in constant time. See Appendix B for the full proof. ◀

The above space bounds can be improved by replacing hashing with any comparison-based in-place integer sorting algorithm:

► **Theorem 5.** *In  $\mathcal{O}(n \log^2 n)$  worst-case time and  $n$  words of space (on top of  $T$ ) we can check whether  $\phi_q$  is collision-free over all pairs of substrings of  $T$  having the same length  $k = 2^e$ , for all  $0 \leq e \leq \log n$*

If we limit the word size to  $w \in \Theta(\log n)$ , then we can use in-place radix sorting [11] to improve the above result:

► **Theorem 6.** *If  $w \in \Theta(\log n)$ , then in  $\mathcal{O}(n \log n)$  expected time and  $n$  words of space (on top of  $T$ ) we can check whether  $\phi_q$  is collision-free over all pairs of substrings of  $T$  having the same length  $k = 2^e$ , for all  $0 \leq e \leq \log n$*

We can now use these results to build a deterministic LCE data structure. We randomly pick pairs  $q, \bar{s}$  and keep re-building our LCE structure until:

1. Its total space usage is of  $n \lceil \log \sigma \rceil$  bits plus a constant number of memory words, and
2.  $\phi_q$  is collision-free over all pairs of substrings of  $T$  having the same length  $k = 2^e$ , for all  $1 \leq e \leq \lfloor \log n \rfloor$

Checking property (1) can be done during construction. As described in Section 3.5, by reversing construction whenever  $|S|$  exceeds some constant chosen in advance, working space never exceeds  $n \lceil \log \sigma \rceil$  bits plus a constant number of memory words. After successful construction, property (2) can be checked with the space/time tradeoffs of Theorems 4-6. We are left to show what is the expected number  $R$  of pairs  $q, \bar{s}$  we have to pick before both properties are satisfied.

**Number of construction rounds** Recall (Section 3.4.1) that  $C$  is the random variable denoting the number of pairs  $\langle X, Y \rangle$  of  $T$  substrings with  $|X| = |Y|$  that generate a collision, i.e.  $X \neq Y$  and  $\phi_q(X) = \phi_q(Y)$ , and that  $S$  is the set containing all positions  $i$  such that the most significant bit of  $P'[i]$  is equal to 1. We are interested in computing a lower bound for the success probability

$$\mathcal{P}(C = 0 \wedge |S| = 0) = 1 - \mathcal{P}(C > 0 \vee |S| > 0) \quad (10)$$

From the inequality  $\mathcal{P}(C > 0 \vee |S| > 0) \leq \mathcal{P}(C > 0) + \mathcal{P}(|S| > 0)$ , we obtain that the quantity in Equation 10 is greater than or equal to

$$1 - \mathcal{P}(C > 0) - \mathcal{P}(|S| > 0) \quad (11)$$

We choose  $\tau = 10w$ . This satisfies Constraints (6), (7), and (8) and implies that—see Section 3.3— $\mathcal{P}(|S| > 0) = 1 - \mathcal{P}(|S| = 0) \leq 1 - 0.5 = 0.5$ . It follows that quantity in Equation 11 is greater than or equal to  $0.5 - \mathcal{P}(C > 0)$ . Finally—see Section 3.4.1—the choice  $\tau = 10w$  implies  $\mathcal{P}(C > 0) \leq n^{-1}$ . This, plugged into the above inequalities, gives us  $\mathcal{P}(C = 0 \wedge |S| = 0) \geq 0.5 - n^{-1}$ . Note that  $n^{-1} \leq 0.25$  holds for  $n \geq 4$ , which is true by assumption. We obtain:  $\mathcal{P}(C = 0 \wedge |S| = 0) \geq 0.25$ . The number  $R$  of rounds of our construction algorithm is a geometric random variable with success probability  $p = \mathcal{P}(C = 0 \wedge |S| = 0) \geq 0.25$ , and has therefore expected value  $1/p \leq 4$ .

We use the technique described in Section 3.4.2 (subsection *Speeding up LCE queries*) in order to compute LCE queries with our structure, so that we only need  $\phi_q$  to be collision-free between text substrings whose lengths are powers of two. If we do not sample values  $2^{\lceil \log \sigma \rceil} \cdot 2^i \bmod q$ ,  $0 \leq i < \log n$ , at each binary/exponential search step we have to compute one of them in  $\mathcal{O}(\log \ell)$  time using the fast exponentiation algorithm. Using the collision-checking procedure described in Theorem 4 we obtain:

► **Theorem 7.** *In  $\mathcal{O}(n \log n)$  expected time and  $\mathcal{O}(n)$  words of space we can build a deterministic data structure taking  $n \lceil \log \sigma \rceil$  bits of space and supporting extraction of any length- $m$  text substring and LCE queries in  $\mathcal{O}(m \log \sigma / w)$  and  $\mathcal{O}(\log^2 \ell)$  worst-case time, respectively. LCE queries can be supported in  $\mathcal{O}(\log \ell)$  time by using  $\mathcal{O}(\log n)$  additional words of space.*

## 5 Applications

In what follows we assume that the memory word's size is  $w = \lceil \log n \rceil$  bits, and that the suffix/LCP arrays are composed by  $n$  such words (or  $b \leq n$  words in the case of the sparse suffix array). This restriction is required in order to being able to in-place radix sort [11]  $\Theta(w)$ -bits fingerprints. As a consequence, our results hold for alphabet size  $\sigma \leq n^{\mathcal{O}(1)}$  (recall that we need  $\sigma \leq 2^{\mathcal{O}(w)}$  in order to manipulate alphabet characters in constant time).

With *slow* LCE queries we denote queries running in  $\mathcal{O}(\log^2 n)$  time and  $\mathcal{O}(1)$  space on top of our LCE data structure. With *fast* LCE queries we denote those running in  $\mathcal{O}(\log n)$  time and requiring  $\tau \cdot \log n = 10w \log n = 10 \log^2 n$  bits of space on top of our LCE data structure (this space is needed to store values  $2^{\lceil \log \sigma \rceil \cdot 2^i} \bmod q$ , see Section 3.4.2).

In our results below, we obtain the space to support fast LCE queries by compressing integer sequences as done in [10, 11]. Given a sequence  $S[1, \dots, k]$  of  $\log n$ -bits integers, with  $k \geq 10 \log^2 n$ , we first sort it in increasing order (in-place and  $\mathcal{O}(k)$  time using in-place radix sort [11]). Then, we store in one word the index of the first integer in the sorted sequence starting with bit '1', and compact  $S$  in  $k \log n - k$  adjacent bits by removing the first bit from each integer. This saves  $k \geq 10 \log^2 n$  bits of space. We store values  $2^{\lceil \log \sigma \rceil \cdot 2^i} \bmod q$  in this space, so that our structure supports fast LCE queries. When fast LCE queries are no more needed,  $S$  can be decompressed and restored in its original (sorted) form.

**Suffix sorting** The lexicographic order of two text suffixes can be easily computed by comparing the two characters following their longest common prefix (i.e. one LCE query and one text access). By carefully combining our Monte Carlo data structure with in-place comparison sorting, in-place merging [31], and compression of integer sequences we obtain:

► **Theorem 8.** *Any set  $S = \{i_1, \dots, i_b\}$  of  $b$  suffixes of a text  $T \in \Sigma^n$ ,  $|\Sigma| \leq n^{\mathcal{O}(1)}$ , can be sorted correctly with high probability in  $\mathcal{O}(n + b \log^2 n)$  expected time using  $\mathcal{O}(1)$  words of space on top of  $T$  and  $S$*

**Proof.** We first build our Monte Carlo structure using the in-place construction algorithm of Section 3.5 ( $\mathcal{O}(n)$  expected time).

If  $b < n/\log^3 n$ , then we suffix-sort the  $b$  text positions plugging slow LCE queries in any in-place comparison sorting algorithm terminating within  $\mathcal{O}(b \log b)$  comparisons. This requires  $\mathcal{O}(n + b \log b \log^2 n) \in \mathcal{O}(n)$  time.

If  $b \geq n/\log^3 n$ , then we divide the array  $S$  of text positions in two sub-arrays  $S' = S[1, \dots, n/\log^3 n]$  and  $S'' = S[n/\log^3 n + 1, \dots, b]$  and:

1. We compress  $S'$  as described at the beginning of this section. This saves  $n/\log^3 n > 10 \log^2 n$  bits of space (note that this inequality holds for  $n$  larger than some constant). We store values  $2^{\lceil \log \sigma \rceil \cdot 2^i} \bmod q$  in this space, so that our structure now supports fast LCE queries.
2. We suffix-sort in-place (comparison-based sorting)  $S''$  using fast LCE queries. This step terminates in  $\mathcal{O}(b \log b \log n) \subseteq \mathcal{O}(b \log^2 n)$  time.
3. We decompress  $S'$ . Now our structure supports only slow LCE queries.
4. We suffix sort in-place (comparison-based sorting)  $S'$  using slow LCE queries. This step terminates in  $\mathcal{O}(n/\log^3 n) \cdot \log(n/\log^3 n) \cdot \log^2 n \in \mathcal{O}(n)$  time
5. We merge  $S'$  and  $S''$  using slow LCE queries and in-place comparison-based merging [31]. This step terminates  $\mathcal{O}(b \log^2 n)$  time.

◀

To the best of our knowledge, Theorem 8 is the first in-place solution for the *sparse suffix sorting* problem improving over the time  $\mathcal{O}(nb)$  achievable using radix sort. Our result improves the space of the state of the art [3, 9, 12, 16, 20]. Note that the above procedure could return a wrong result with small probability. If we wish to compute the full suffix array, however, we can de-randomize our structure with Theorem 5 before applying Theorem 8 to the sequence  $S = \{1, \dots, n\}$ . We obtain:

► **Theorem 9.** *The suffix array  $SA$  of  $T \in \Sigma^n$ ,  $|\Sigma| \leq n^{\mathcal{O}(1)}$ , can be computed in  $\mathcal{O}(n \log^2 n)$  expected time using  $\mathcal{O}(1)$  words of space on top of  $T$  and  $SA$ .*

Theorem 9 does not improve the state of the art [10]. However, it represents the only other known solution to the problem so we think it is worth mentioning.

**LCP array construction** the *Longest Common Prefix* (LCP) array stores the LCEs of lexicographically consecutive suffixes, and is often used to speed up search algorithms based on the suffix array and to simulate the suffix tree. To date, the most space-efficient and practical LCP construction algorithms [7, 13, 19, 28] rely on succinct and compressed data structures, and use  $\mathcal{O}(n)$  bits of auxiliary storage on top of the text and the LCP array. The previous fastest in-place LCP array construction algorithm [26] runs in quadratic time, and is therefore not practical on big texts. Here we present the first practical in-place LCP construction algorithm. The result, stated in the following theorem, follows from a careful combination of our deterministic LCE data structure, in-place suffix sorting [10], in-place radix sorting [11], and compression of increasing sequences of integers:

► **Theorem 10.** *The Longest Common Prefix array (LCP) of  $T \in \Sigma^n$ ,  $|\Sigma| \leq n^{\mathcal{O}(1)}$ , can be computed in  $\mathcal{O}(n \log n)$  expected time using  $\mathcal{O}(1)$  words of space on top of the text and the LCP.*

**Proof.** Our procedure consists of the following steps:

1. We build and de-randomize our LCE structure using Theorem 6. We store in two words the modulo  $q$  and the seed  $\bar{s}$  computed during construction and re-store the text.
2. We build the suffix array  $SA$  in-place and  $\mathcal{O}(n \log n)$  time using [10].
3. We re-build our deterministic LCE structure using the values  $q$  and  $\bar{s}$  computed in step (1). This step runs in-place and  $\mathcal{O}(n)$  time.
4. We compress the integers in  $SA' = SA[1, \dots, n/\log^2 n]$  with the procedure described at the beginning of this section. This saves  $n/\log^2 n > 10 \log^2 n$  bits of space (this inequality holds for  $n$  larger than some constant). We store values  $2^{\lceil \log \sigma \rceil \cdot 2^i} \bmod q$  in this space, so that our structure now supports fast LCE queries. Note that  $SA'$  is no more suffix-sorted.
5. We convert  $SA'' = SA[n/\log^2 n + 1, \dots, n]$  to  $LCP[n/\log^2 n + 1, \dots, n]$  by computing LCE values of adjacent suffixes using fast LCE queries. This step runs in-place and  $\mathcal{O}(n \log n)$  time.
6. We decompress  $SA'$ . Now our LCE structure supports only slow LCE queries.
7. We suffix-sort in-place (comparison-based sorting)  $SA'$  using slow LCE queries. This step runs in-place and  $\mathcal{O}((n/\log^2 n) \log(n/\log^2 n) \log^2 n) = \mathcal{O}(n \log n)$  time.
8. We convert  $SA'$  to  $LCP[1, \dots, n/\log^2 n]$  by computing LCE values of adjacent suffixes using slow LCE queries. This step runs in-place and  $\mathcal{O}((n/\log^2 n) \log^2 n) = \mathcal{O}(n)$  time. ◀

---

## References

- 1 Jean-Paul Allouche, Michael Baake, Julien Cassaigne, and David Damanik. Palindrome complexity. *Theoretical Computer Science*, 292(1):9–31, 2003.

- 2 Amihoud Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with  $k$  mismatches. *Journal of Algorithms*, 50(2):257–275, 2004.
- 3 Philip Bille, Johannes Fischer, Inge Li Gørtz, Tsvi Kopelowitz, Benjamin Sach, and Hjalte Wedel Vildhøj. Sparse text indexing in small space. *ACM Transactions on Algorithms (TALG)*, 12(3):39, 2016.
- 4 Philip Bille, Inge Li Gørtz, Mathias Bæk Tejs Knudsen, Moshe Lewenstein, and Hjalte Wedel Vildhøj. Longest common extensions in sublinear space. In *Annual Symposium on Combinatorial Pattern Matching*, pages 65–76. Springer, 2015.
- 5 Philip Bille, Inge Li Gørtz, Benjamin Sach, and Hjalte Wedel Vildhøj. Time–space trade-offs for longest common extensions. *Journal of Discrete Algorithms*, 25:42–50, 2014.
- 6 Dany Breslauer and Zvi Galil. Finding all periods and initial palindromes of a string in parallel. *Algorithmica*, 14(4):355–366, 1995.
- 7 Johannes Fischer. Inducing the LCP-array. In *Workshop on Algorithms and Data Structures*, pages 374–385. Springer, 2011.
- 8 Johannes Fischer and Volker Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Annual Symposium on Combinatorial Pattern Matching*, pages 36–48. Springer, 2006.
- 9 Johannes Fischer, I Tomohiro, and Dominik Köppl. Deterministic sparse suffix sorting on rewritable texts. In *Latin American Symposium on Theoretical Informatics*, pages 483–496. Springer, 2016.
- 10 Gianni Franceschini and S Muthukrishnan. In-place suffix sorting. In *International Colloquium on Automata, Languages, and Programming*, pages 533–545. Springer, 2007.
- 11 Gianni Franceschini, S. Muthukrishnan, and Mihai Patrascu. Radix sorting with no extra space. In *Proc. 15th ESA*, pages 194–205, 2007.
- 12 Paweł Gawrychowski and Tomasz Kociumaka. Sparse suffix tree construction in optimal time and space. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 425–439. SIAM, 2017.
- 13 Simon Gog and Enno Ohlebusch. Fast and lightweight LCP-array construction algorithms. In *Proceedings of the Meeting on Algorithm Engineering & Experiments*, pages 25–34. Society for Industrial and Applied Mathematics, 2011.
- 14 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355, 1984.
- 15 DR Heath-Brown. The differences between consecutive primes. *Journal of the London Mathematical Society*, 2(1):7–13, 1978.
- 16 Tomohiro I, Juha Kärkkäinen, and Dominik Kempa. Faster Sparse Suffix Sorting. In *31st International Symposium on Theoretical Aspects of Computer Science*, pages 386–396. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.
- 17 Shunsuke Inenaga. A faster longest common extension algorithm on compressed strings and its applications. In *Prague Stringology Conference*, pages 1–4, 2015.
- 18 Marc Joye and Pascal Paillier. Fast generation of prime numbers on portable devices: An update. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 160–173. Springer, 2006.
- 19 Juha Kärkkäinen, Giovanni Manzini, and Simon J Puglisi. Permuted longest-common-prefix array. In *Annual Symposium on Combinatorial Pattern Matching*, pages 181–192. Springer, 2009.
- 20 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006.
- 21 Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

- 22 Roman Kolpakov, Mikhail Podolskiy, Mikhail Posypkin, and Nickolay Khrapov. Searching of gapped repeats and subrepetitions in a word. In *Symposium on Combinatorial Pattern Matching*, pages 212–221. Springer, 2014.
- 23 Dmitry Kosolobov. Tight lower bounds for the longest common extension problem. *arXiv preprint arXiv:1611.02891*, 2016.
- 24 Gad M Landau, Jeanette P Schmidt, and Dina Sokol. An algorithm for approximate tandem repeats. *Journal of Computational Biology*, 8(1):1–18, 2001.
- 25 Gad M Landau and Uzi Vishkin. Fast parallel and serial approximate string matching. *Journal of algorithms*, 10(2):157–169, 1989.
- 26 Felipe A Louza and Guilherme P Telles. Computing the BWT and the LCP Array in Constant Space. In *International Workshop on Combinatorial Algorithms*, pages 312–320. Springer, 2015.
- 27 Helmut Maier et al. Primes in short intervals. *The Michigan Mathematical Journal*, 32(2):221–225, 1985.
- 28 Giovanni Manzini. Two space saving tricks for linear time LCP array computation. In *Scandinavian Workshop on Algorithm Theory*, pages 372–383. Springer, 2004.
- 29 Eugene W Myers. An  $O(ND)$  difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.
- 30 Takaaki Nishimoto, I Tomohiro, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully Dynamic Data Structure for LCE Queries in Compressed Space. In *41st International Symposium on Mathematical Foundations of Computer Science*, pages 72:1–72:15, 2015.
- 31 Jeffrey S. Salowe and William L. Steiger. Simplified stable merging tasks. *J. Algorithms*, 8(4):557–571, 1987.
- 32 Yuka Tanimura, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, Simon J. Puglisi, and Masayuki Takeda. Deterministic sub-linear space LCE data structures with efficient construction. In *Annual Symposium on Combinatorial Pattern Matching*. Springer, 2016.

## **A** Simulating larger word sizes

Since we make use only of integer additions, multiplications, modulo, and bitwise operations (masks, shifts), we can assume that we can simulate a memory word of size  $w' = c \cdot w$  for any constant  $c$  with only a constant slowdown in the execution of these operations. Subtractions, additions and multiplications between  $(c \cdot w)$ -bits words take trivially constant time by breaking the operands in  $2c$  digits of  $w/2$  bits each and use schoolbook’s algorithms (i.e.  $\mathcal{O}(c)$ -time addition and  $\mathcal{O}(c^2)$ -time multiplication). The modulo operator  $a \bmod q$  (with  $q$  fixed) can be computed as  $a \bmod q = a - \lfloor a/q \rfloor \cdot q$ . Computing  $\lfloor a/q \rfloor$  requires pre-computing (once) the floating point value  $q^{-1}$  up to some fixed precision (i.e.  $\mathcal{O}(w)$  decimal binary digits since  $a$  and  $q$  have  $\mathcal{O}(w)$  binary digits), performing a (constant time) multiplication  $a \cdot q^{-1}$ , and truncating the result to an integer. Bitwise operations on  $(c \cdot w)$ -bits words can trivially be implemented with  $c$  bitwise operations between  $w$ -bits words.

## **B** Appendix: proofs

### **B.0.1** Proof of Theorem 4

The idea is to check the property on strings of length  $2^e$  by using the already-checked property on strings of length  $2^{e-1}$ . First, we build a data structure supporting the computation of  $\phi_q(T[i, \dots, j])$  in constant time. To this end, we can use the structure described in the previous section, augmented with  $n$  words storing values  $2^{\lceil \log n \rceil \cdot i} \bmod q$ ,  $i = 0, \dots, n-1$  (to guarantee constant-time retrieval of powers of 2 modulo  $q$ ). We start with  $e = 0$  and repeat



$\lceil \log n \rceil + 1$  times the following procedure, each time incrementing  $e$  by one. We use a hash table  $\mathcal{H}$  of size  $\mathcal{O}(n)$  with associated hash function  $h : [0, q-1] \rightarrow [0, |\mathcal{H}|-1]$  mapping Karp-Rabin fingerprints of length- $2^e$  text substrings to numbers in  $[0, |\mathcal{H}|-1]$ .  $\mathcal{H}$  can be implemented, e.g. with linear probing in order to guarantee expected constant-time operations. Each entry of  $\mathcal{H}$  is associated with a list of integers. We scan  $T$  left-to-right and, for each  $0 \leq i \leq n-2^e$ , append the value  $i$  at the end of the list  $\mathcal{H}[h(\phi_q(T[i, \dots, i+2^e-1]))]$ . Then, for each  $0 \leq t < |\mathcal{H}|$ , we check that all  $i_1, i_2 \in \mathcal{H}[t]$  are such that  $T[i_1, \dots, i_1+2^e-1] = T[i_2, \dots, i_2+2^e-1]$ . This task can be performed in  $\mathcal{O}(|\mathcal{H}[t]|)$  time as follows.

Let  $\mathcal{H}[t] = \langle i_0, \dots, i_{d-1} \rangle$ . We only need to perform  $d-1$  comparisons  $T[i_j, \dots, i_j+2^e-1] = T[i_{j+1}, \dots, i_{j+1}+2^e-1]$  for  $0 \leq j < d$ . If  $e = 0$ , then each comparison takes constant time and can be done by simply accessing the text. If  $e > 0$ , then  $T[i_j, \dots, i_j+2^e-1] = T[i_{j+1}, \dots, i_{j+1}+2^e-1]$  holds if and only if both  $\phi_q(T[i_j, \dots, i_j+2^{e-1}-1]) = \phi_q(T[i_{j+1}, \dots, i_{j+1}+2^{e-1}-1])$  and  $\phi_q(T[i_j+2^{e-1}, \dots, i_j+2^e-1]) = \phi_q(T[i_{j+1}+2^{e-1}, \dots, i_{j+1}+2^e-1])$  hold (constant time by using our structure to compute any  $\phi_q(T[i', \dots, j'])$ ). Note that we already verified that  $\phi_q$  is collision-free over  $T$ 's substrings of length  $2^{e-1}$ , so both checks are deterministic. All lists in  $\mathcal{H}$  store overall  $n-2^e+1$  elements, therefore the procedure terminates in  $\mathcal{O}(n)$  expected time. Since we have to repeat this for every non-negative integer  $e \leq \log n$ , the overall expected time is  $\mathcal{O}(n \log n)$ . ◀

## B.0.2 Proof of Theorem 5

First, we build in-place and  $\mathcal{O}(n)$  time the in-place data structure supporting the computation of  $\phi_q(T[i, \dots, j])$  in  $\mathcal{O}(\log n)$  time described in Section 3.5.

For  $e = 0, \dots, \lceil \log n \rceil$  we repeat the following procedure. We initialize an array (text positions)  $A[0, \dots, n-2^e]$  with  $A[i] = i$ . We use any  $\mathcal{O}(n \log n)$  in-place comparison-sorting algorithm to sort  $A$  according to length- $2^e$  fingerprints, i.e. using the ordering  $\prec$  defined by  $A[i] \prec A[j]$  iff  $\phi_q(T[A[i], \dots, A[i]+2^e-1]) < \phi_q(T[A[j], \dots, A[j]+2^e-1])$ . At this point, we scan  $A$  and, for every pair of adjacent  $A$ 's elements  $A[i], A[i+1]$ , if  $\phi_q(T[A[i], \dots, A[i]+2^e-1]) = \phi_q(T[A[i+1], \dots, A[i+1]+2^e-1])$ , then we check deterministically that the two substrings  $T[A[i], \dots, A[i]+2^e-1]$  and  $T[A[i+1], \dots, A[i+1]+2^e-1]$  are indeed equal with the same strategy used in the proof of Theorem 4 (i.e. we compare the fingerprints of their two halves of length  $2^{e-1}$ , or we just access the text if  $e = 0$ ). Finally, we free the memory allocated for  $A$ .

Analysis. For every  $e \leq \log n$  we sort  $A$  ( $\mathcal{O}(n \log n)$  comparisons). Note that fingerprints have all the same length  $2^e$ , so we only need to sample value  $2^{b \cdot 2^e} \bmod q$ , with  $b = \lceil \log \sigma \rceil$ , in order to speed up fingerprint computation (see Section 3.4.2—*Speeding up LCE queries*). Each comparison in the sorting algorithm requires the computation of two fingerprints and takes therefore constant time. We moreover need value  $2^{b \cdot 2^{e-1}} \bmod q$  to perform the deterministic collision check. Since  $2^{b \cdot 2^e} \bmod q$  can be computed in constant time from  $2^{b \cdot 2^{e-1}} \bmod q$ , we need to reserve only two memory words for this sampling of powers of 2 modulo  $q$  (updating these two values every time  $e$  is incremented). ◀

## B.0.3 Proof of Theorem 6

We need to show that we can sort in-place (i.e.  $n \log n$  bits of space) and  $\mathcal{O}(n)$  time text positions  $i = 0, \dots, n-2^e$  using as comparison values  $\phi_q(T[i, \dots, i+2^e-1])$ . Then, we plug this sorting procedure in the proof of Theorem 5 to obtain the claimed bounds of the theorem.



Let  $w = c \cdot \log n$ , with  $c \in \mathcal{O}(1)$ . A Karp-Rabin fingerprint takes  $\tau = 10w = 10c \log n$  bits of space. Let  $x_i = \phi_q(T[i, \dots, i + 2^e - 1])i$  be the concatenation of the fingerprint of  $T[i, \dots, i + 2^e - 1]$  and of position  $i$  written in binary.  $x_i$  takes  $(10c + 1) \log n$  bits of space (note that Karp-Rabin Fingerprints can be computed in constant time using our Monte Carlo structure as the power  $2^e \bmod q$  is fixed). We store  $x_0, \dots, x_{n/(10c+1)-1}$  in an array  $A$  taking  $n \log n$  bits of space. We sort  $x_0, \dots, x_{n/(10c+1)-1}$  in-place and  $\mathcal{O}(n)$  time using in-place radix sort [11]. Then, we compact  $A$  by replacing each  $x_i$  with the integer  $i$ . As a result, the first  $n/(10c + 1)$  entries of  $A$  now contain text positions  $0, \dots, n/(10c + 1) - 1$  sorted by their fingerprint. We apply recursively the above procedure to text positions  $n/(10c + 1), \dots, n - 1$  using the free space left in  $A$  (i.e.  $n \log n - \frac{n}{10c+1} \log n$  bits) to perform sorting. Note that we recurse on  $n - \frac{n}{10c+1} = n \frac{10c}{10c+1}$  text positions. Let  $d = \frac{10c+1}{10c} > 1$ . This gives us the recurrence  $T(n) = \mathcal{O}(n) + T(n/d)$  for our overall procedure (with base case  $T(1) = \mathcal{O}(1)$ ), which results in overall  $T(n) = \mathcal{O}(n)$  time (being  $d > 1$ ).

After terminating the above procedure,  $A$  contains  $\mathcal{O}(\log n)$  sub-arrays of text positions sorted by their fingerprints. Starting from the two rightmost such sub-arrays, we repeatedly apply in-place merge sort [31] until the whole  $A$  is sorted. Note that a single comparison of two text positions  $i$  and  $j$  requires computing  $\phi_q(T[i, \dots, i + 2^e - 1])$  and  $\phi_q(T[j, \dots, j + 2^e - 1])$  (constant time with our structure and using the sampled value  $2^e \bmod q$ ). Boundaries of the sub-arrays can be computed on-the-fly (i.e.  $0, n - n/d, n - n/d^2, \dots$ ). Analogously to the above analysis, at the  $j$ -th step,  $j \geq 1$ , we merge in linear time two sub-arrays of total size  $\mathcal{O}(d^j)$ . The overall time spent inside this procedure is therefore  $\mathcal{O}(\sum_{i=1}^{\log_d n} d^i) = \mathcal{O}(n)$ .

#### B.0.4 Acknowledgments

I wish to thank Tomasz Kociumaka for reviewing the first draft of the paper and spotting some imprecisions in the proofs, and anonymous reviewers for useful comments on a preliminary version of the paper.